**Writing a Snake game**

**in AQA Assembly Language**

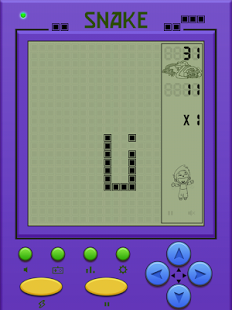
STUDENT WORKBOOK

Created by Richard Pawson, Stowe School

[Creative Commons License](https://creativecommons.org/licenses/by-sa/4.0/)  
This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

# Introduction

Most people who play video games will have played ‘Snake’ or ‘Serpant’ at some point: steering a snake around the screen to gobble apples, the snake growing one segment longer for each apple eaten. More sophisticated versions of the game exist for modern PCs and mobile phones, but the earliest versions were created for hand-held dedicated game devices, with low-resolution monochrome liquid-crystal displays, like the one shown below:

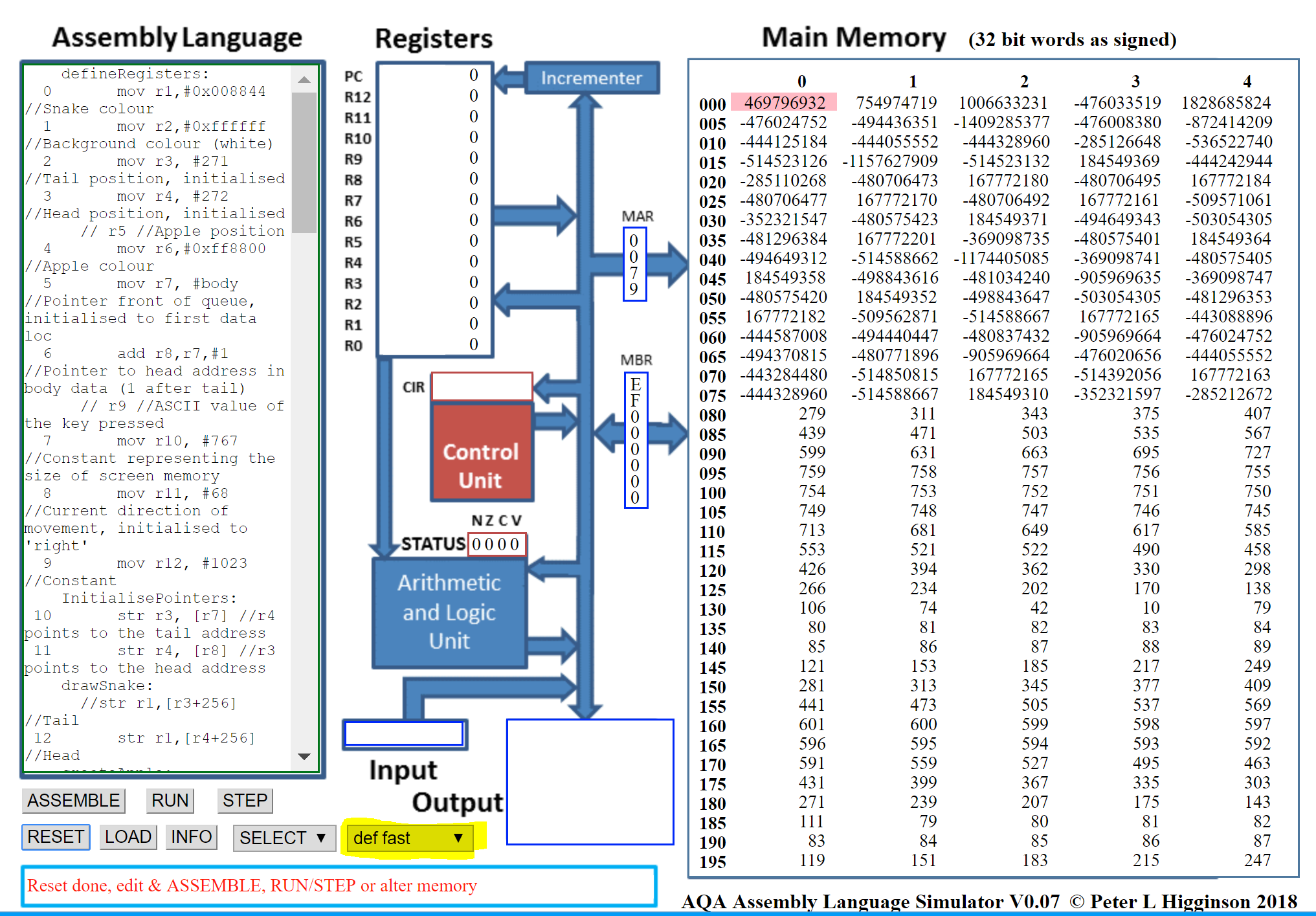


The earliest versions of these devices were built from 8-bit microprocessors, with very limited memory. Both for performance and memory efficiency, the software for this and similar games was written in assembly language and then assembled into machine code.

In this workbook we are going to re-create a simple version of Snake from scratch, written in a modern assembly language to run on a processor simulation. With very little modification the same code could run on, say, a Raspberry Pi with a memory-addressable display device.

# Using the simulation

We shall be using the AQA [processor simulation](http://www.peterhigginson.co.uk/AQAY/) written by Peter Higginson as illustrated below:



Before you start, set the simulation options to **def fast** as highlighted in the image above. This is the fastest mode of execution - otherwise the simulation will animate the workings of the processor, which is interesting to observe but results in a less exciting game.

This simulator is an attempt to implement the specification of the AQA Assembly Language. Unfortunately the AQA specification is not complete, so some interpretation and filling in has been necessary. The simulated processor has much in common with the architecture of the ARM processor, but it is not identical.

### Addressing modes

The AQA specification for assembly language covers only the *immediate* and *direct* addressing modes: it does not cover *indexed*, or *indirect* addressing modes – even though the ARM processor (from which the AQA specification has clearly been derived) does have those modes, as do most processors, old and new. In fact, you can’t write any interesting programs in assembly language without either indexed or indirect modes, unless you resort to self-modifying programs, which is not considered good practice and which many modern computers prevent for security reasons. The simulator does include indexed and indirect addressing modes, and these are both used in the Snake game.

We believe it better that you learn all four of the *main* addressing modes (some processors offer even more) – immediate, direct, indexed, and indirect. But remember, in answering an AQA exam question, that the AQA spec covers only the first two.

# The development process

In this workbook, development of the Snake game is broken up into a series of small stories each of delivers a simple working system that you can test. Within a few stories the application will start to resemble the real game. This approach, known as *agile development* is roughly how modern systems development is managed, even for very large projects.

Each story requires no more than 10 machine instructions to be added, or existing ones modified. The final version of the game is less than 90 instructions.

We recommended that you write and modify your code in a text editor - which can be as simple as NotePad. When you have made the code changes needed for a given story in the text editor, save the new version and then copy and paste the complete code for the story into the simulator - ready to Submit and Run. Editing the code directly within the Assembly Language pane on the simulator is possible, but less convenient. (If you store your code in a *code versioning system* such as Git, that’s even better).

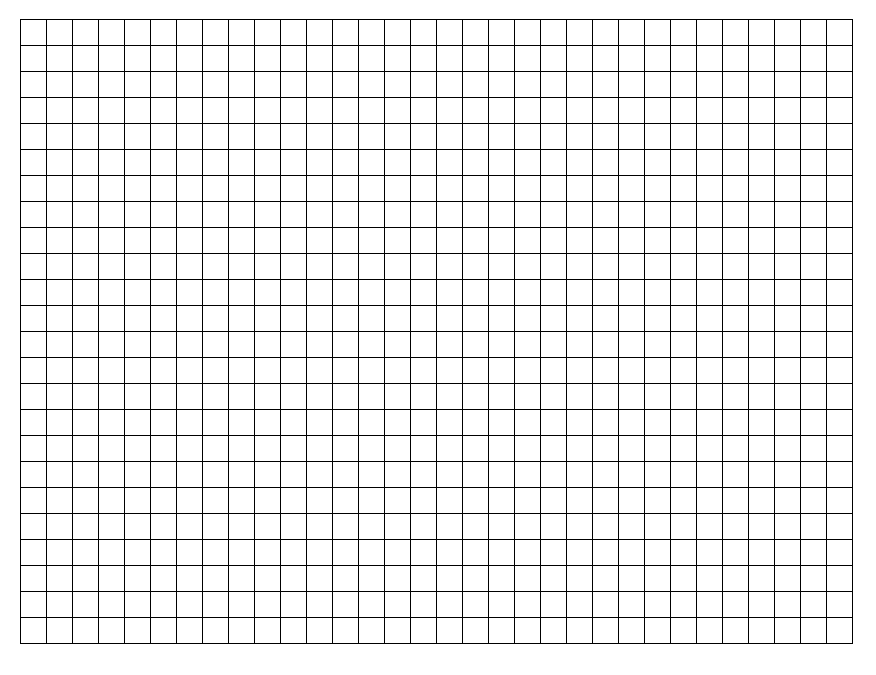
# Story 1: Draw the starting snake and apple

## Requirement

Create a two-segment snake (head + tail) near the middle of the screen. The snake should be a green colour. Add an apple (a single pixel of a different colour) a little below the snake.

## Algorithm & techniques

The simulation provides ‘addressable video memory’, that runs from (word) location 256 (0x100 in hex) for the top-left corner as shown below:



Address 256

287

1023

32 columns

24

rows

mov r1,#0 means ‘move into register 1, the value 0’. This is known as ‘immediate address mode’. In this case, the value 0 represents the colour black (no colour).

When the program starts, the simulation will set the value of all registers to the default value 0. However, on a real processor it is not safe to assume this, hence we set the value of register 0 to 0 explicitly. This is equivalent to the practice - in a higher level language - of initialising all variables.

We can then store this value into the address corresponding to the screen pixel we want, for example:

mov r1,#0

str r1,256

In the above code, 256 does not mean ‘use the (immediate) value of 256’ - the instruction means ‘store the contents of register 1 in memory location 256’. This is an example of ‘direct address mode’.

Try this out for yourself. Then try adding similar instructions to paint each of the four corner squares black.

Paste a partial screenshot showing the simulations Output window with the four corner squares painted black.

We can specify other colours using the same RGB (Red Green Blue) format as used when creating a web page. This is best specified in hex, so for example, 0x008844 results in a suitable hue of green for the snake.

Try altering the code above to use different colour values, specified in decimal or hex. Paste in the result.

We’re now ready to write the code for the first story.

## Code changes

defineRegisters:

mov r1,#0x008844 //Snake colour (green)

drawSnake:

str r1,527 //Tail

str r1,528 //Head

Load the file into the simulator, assemble and run (making sure you have set the simulation to ‘def fast’ mode).

Paste in a partial screenshot showing the assembled code and the Output after running.

Notice that we have added two labels: defineRegisters: and drawSnake:. These aren’t actually used by the program at this point, but they make the code more readable.

Why have we used r1 rather than r0? It is a common convention, though my no means necessary, to keep r0 for storing temporary values and calculations.

# Story 2: Move the snake

## Requirement

As soon as the snake has been drawn, it should start moving to the right, one pixel at a time, continuously.

## Algorithm & techniques

To move the snake one pixel to the right we *could* draw a new head in the next screen memory location (529) and then reset the tail (527) to the background colour (white) something like this (don’t make these changes to your code):

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

drawSnake:

str r1,527 //Tail

str r1,528 //Head

moveSnake:

str r2,527 //Restore tail location to background colour

str r1,529 //Add head in new position

The problem with this approach is that it doesn’t *generalise*. We will have to add two new instructions for each pixel that the snake moves, and we won’t be able to vary it (eventually) based on live directions from the player.

So, instead, we are going to *refactor* the code from story 1, making use of two more registers to hold the position of the head and tail:

mov r3, #528 //Tail position, initialised

mov r4, #527 //Head position, initialised

And then we are going to use these registers in the drawSnake routine, using another form of addressing called *indirect addressing mode*, signalled by square brackets:

drawSnake:  
 str r1,[r3] //Tail  
 str r1,[r4] //Head

The first instruction above can be read as ‘store the value held in r1 (the snake colour) into the memory address that is held in r4 (i.e. initially, memory location 527). *Indirection* lies at the heart of many advanced programming Algorithm & techniques.

Having done this, we can adjust the values held in r3 and r4 to point to new locations and then use the same store instructions to re-draw it. And if we do this in a loop then we can move the snake continuously to the right.

## Code changes

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

mov r3, #527 //Tail position, initialised

mov r4, #528 //Head position, initialised

drawSnake:

str r1,[r3] //Tail

str r1,[r4] //Head

moveSnake:

str r2,[r3] //Reset tail to Background

add r3,r3,#1 //Increment the tail pointer by 1

add r4,r4,#1 //Increment the head pointer by 1

str r1,[r4] //Draw new head

b moveSnake //Loop

Modify your code from Story 1 (all new lines, and changes to existing lines, are highlighted).

Copy the code into the simulation, assemble, and run.

What happens when the snake gets to the right hand edge of the screen area, and why?

If you leave the program to run long enough you will get an error. On which instruction number has the error occurred? (Instruction numbers are added automatically by the simulator) when the code is assembled)

Which register is being used in that instruction, and what value is it holding at that time?

Why does this cause an error?

# Story 3: Refactor to use indexed addressing

## Requirement

This story does not add any value to the user. It is an example of *refactoring* – which means ‘improving the design of your code without changing functionality’. The motivation for refactoring may be to make it easier to change/extend the design in future (in response to new requirements), to make it more efficient in execution, or simply to make the code easier to read.

## Algorithm & techniques

We are going to make use of another new addressing mode known as *indexed addressing mode* whereby the address is calculated from a base address plus a variable amount (the index value) that is held in a register.

We learned in Story 1 that screen memory runs from locations 256 to 1023. So far we have set the position of the snake *absolutely* e.g. using 528 as the starting position of the head, somewhere near the middle of the screen.

It would be more elegant, and make position-related calculations easier, if we could make all such positions *relative* to the start of screen memory. Thus, location 528 would be replaced by 272 (528 = 256 + 272). The assembler allows us to do this using *indexed* addressing mode. For example, the instruction:

str r1,[r3+256]

Will store the contents of r1 into the address defined by the ‘base address’ 256 plus whatever value is held in r3 (known as the ‘index’ for this purpose).

## Code changes

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

mov r3, #271 //Tail position, initialised

mov r4, #272 //Head position, initialised

drawSnake:

str r1,[r3+256] //Tail

str r1,[r4+256] //Head

moveSnake:

str r2,[r3+256] //Reset tail to Background

add r3,r3,#1 //Increment the tail pointer by 1

add r4,r4,#1 //Increment the head pointer by 1

str r1,[r4+256] //Draw new head

b moveSnake //Loop

# Story 4: Add an apple, and allow the snake to eat it

## Requirement

At the start of the game, draw an apple (one pixel of a different colour) in a position a little below the starting point of the snake. (In a later story we will want to position the apple randomly.) When the snake passes over the apple, the apple should disappear.

## Algorithm & techniques

We have already learned the patterns we need, so we can go straight to implementation

## Code changes

Make the changes highlighted below.

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

mov r3, #271 //Tail position, initialised

mov r4, #272 //Head position, initialised

mov r5, #520 //Apple position

mov r6, #0xff8800 //Apple colour

drawSnake: (unchanged)

moveSnake:

str r6,[r5+256] //Draw Apple each cycle, in case it is on snake

str r2,[r3+256] //Reset tail to Background

When you run the game, and the snake has passed over the apple, what happens to the apple, and why?

# Story 5: Grow the snake

## Requirement

When the snake passes over the apple, the snake’s length should grow by one.

## Algorithm & techniques

We can detect the event of ‘eating’ the apple, by checking, within the loop, when the snake’s head position matches that of the apple. We can the grow the snake length by one, simply by not updating the position of the tail for that cycle of the loop.

Now we need to introduce conditional branching (or ‘selection’) into our code, and we can do this using the cmp (compare) and beq (branch if equals) instructions. We will also use the add instruction to increment a register value by 1.

If you have not previously used these instructions, look up the meaning and the syntax of these instructions on the AQA Assembly Language instruction sheet.

## Code changes

defineRegisters: (unchanged)

drawSnake: (unchanged)

moveSnake:

str r6,[r5+256] //Draw Apple each cycle, in case it is on snake

add r4,r4,#1 //Increment the head location by 1

cmp r4,r5 //If the head is in same location as apple...

beq moveHead //...Skip updating the tail, to make snake grow

moveTail:

str r2,[r3+256] //Reset tail to Background

add r3,r3,#1 //Increment the tail pointer by 1

moveHead:

~~add r4,r4,#1~~ DELETE THIS LINE

str r1,[r4+256] //Draw new head

b moveSnake //Loop

Check that the snake does indeed grow when it passes over the apple. But what happens to the apple, and why?

Having made the modifications and tested that it works, *temporarily*  add code to draw two further apples lower down the screen. Test that the snake grows with each apple eaten, and paste a screenshot of the Output area showing the snake having grown to 5 segments.

Then remove your additional apples - go back to just one.

# Story 6: Change of direction

## Requirement

When the S key is pressed, the snake should switch to moving downwards. The snake should continue downwards until its head reaches the bottom of the screen.

## Algorithm & techniques

The following code probably looks like it *should* work, but it actually contains a subtle bug.

moveSnake:

str r6,[r5+256] //Draw Apple each cycle, in case it is on snake

inp r0,4

cmp r0,#83 //S key

beq down

right:

mov r0,#1 //Re-purpose r0 to hold an increment of 1

b reDraw

down:

mov r0,#32 //32 moves down one row on screen

reDraw:

add r4,r4,r0 //Increment the head pointer by value of r0

cmp r4,r5 //If the head is in same location as apple...

beq moveHead //...Skip updating the tail, to make snake grow

str r2,[r3+256] //Reset tail to Background

add r3,r3,r0 //Increment the tail pointer by value of r0

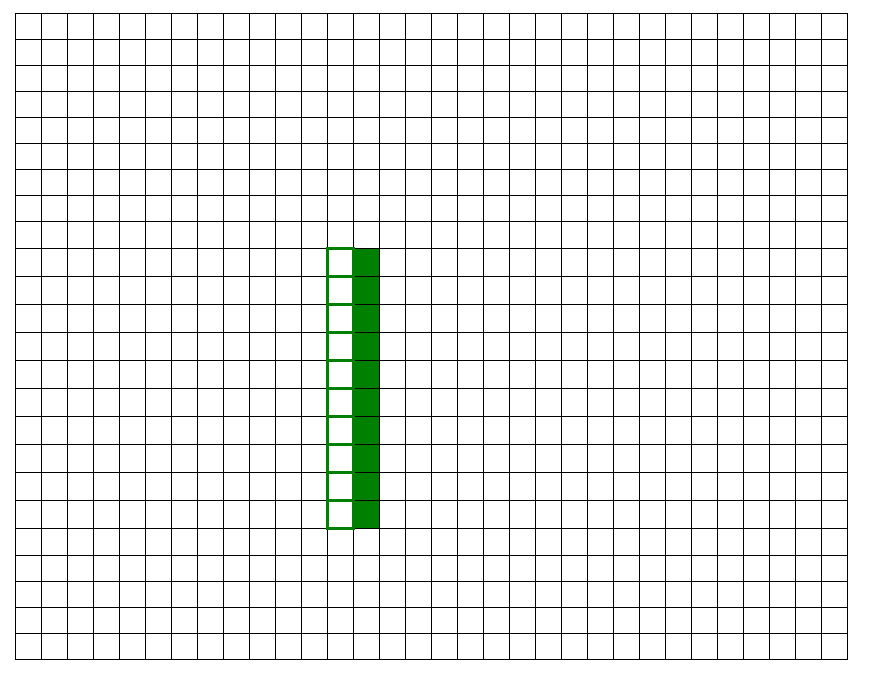
str r1,[r4+256] //Draw new head

b moveSnake //Loop

**What happens when you run the game and hit the ‘s’ key to change direction, downwards?**

**Try stepping through the program, hitting the ‘s’ key early to see if you can see what the bug is?**

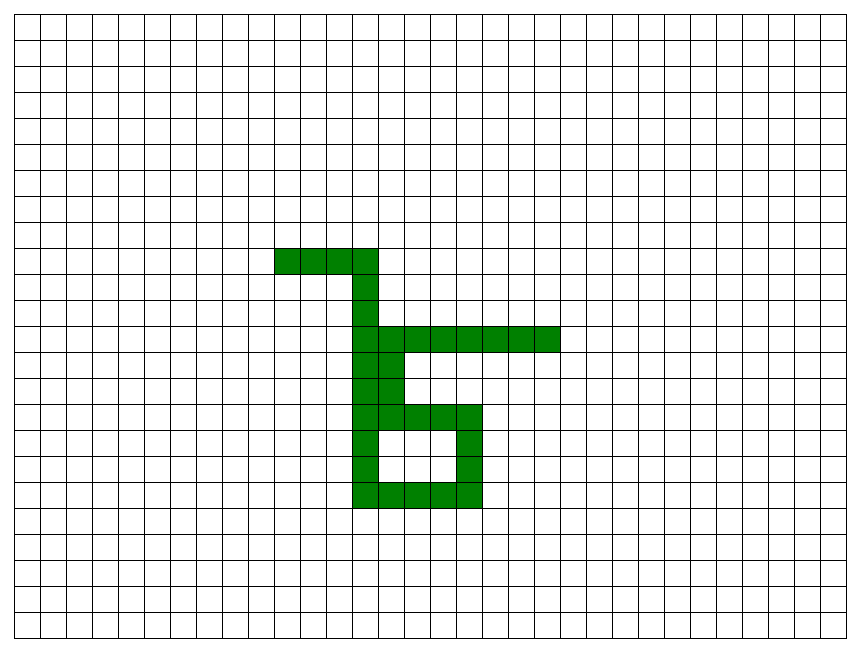
The bug is quite subtle. Upon the change in direction we immediately start moving both the head and the tail locations (held in r3 and r4) downwards. But because the tail is (initially) one to the left of the head, they are going to move down parallel to each other, not in the same column of pixels. So the new trail created by the advancing head is never reset to the background colour by the advancing tail:



And were you to change direction *after* the snake has eaten the apple and grown to three segments, the head and tail will move downwards in parallel even further apart.

To fix this properly, and, especially, to cope with later versions of the game where the snake may acquire a complex shape from many turns, we really need to ensure that the tail always follows the same path as the head, lagging behind by as many segments as the snake is long.

Could this be done by getting the tail-update routine to read the screen memory, looking to see in which direction is the next snake-coloured pixel. That might work for simple cases, but won’t work if the snake has grown longer and doubled-back on itself e.g.



The proper solution is to keep a record of locations the head has passed through, elsewhere in memory. So, before we can continue, we need to do another refactoring.

Before moving onto the refactoring (Story 7), undo the changes made in this story i.e. revert to the code at the end of story 5.

Being willing to ‘back track’ in your coding is an essential part of good coding practice. Don’t be tempted to hold onto the wrong code just because you spent time writing it!

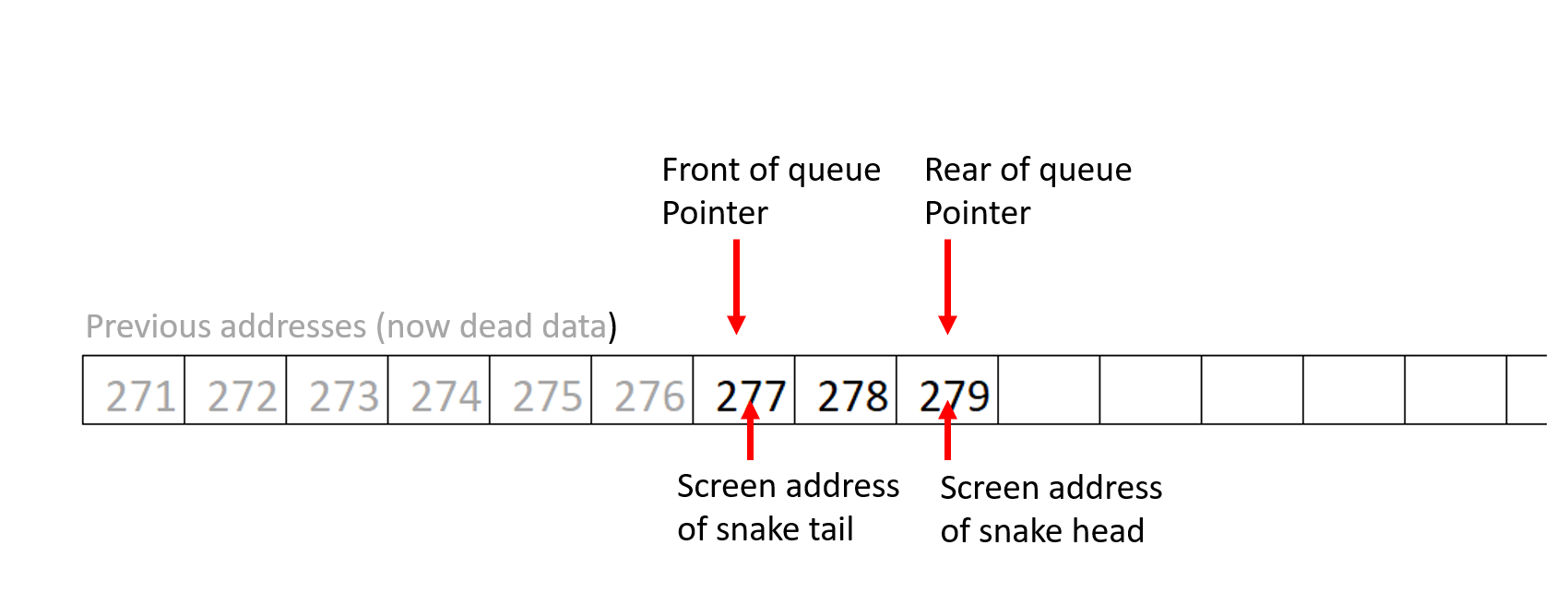
# Story 7: Refactoring

## Requirement

Without changing the functioning of the game (from Story 5), change the implementation such that it maintains a record of the positing of the snake’s head in a data structure within main memory.

## Algorithm & techniques

We can create a ‘queue’, which you should be familiar with from studying Data Structures. We will have two pointers, pointing to the front and back of the data in the queue, corresponding to the position of the tail and head respectively. That might sound the wrong way round, but think about it: each time the head moves we will *enqueue* a new value (the head’s new location) i.e. add it to the back of the queue and change the pointer; and each time the tail moves we will *dequeue* (and discard) the value at the front of the queue, updating the front pointer:



We will define all the memory from the end of the program as being available for the queue using dat as follows:

body: dat 0 //body segment pointers extend from here to end of memory (addr 199)

## Code changes

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

mov r3, #271 //Tail position, initialised

mov r4, #272 //Head position, initialised

mov r5, #330 //Apple position

mov r6,#0xff8800 /Apple colour

mov r7, #body //Pointer front of queue

add r8,r7,#1 //Pointer to head address

InitialisePointers:

str r3, [r7] //r4 points to the tail address

str r4, [r8] //r3 points to the head address

drawSnake: (unchanged)

moveSnake: (unchanged)

moveTail:

ldr r0, [r7]

str r2,[r0+256] //Reset tail to Background

add r7,r7,#1 //Increment the tail pointer (for use next cycle)

moveHead:

add r8,r8,#1 //Increment the head pointer

str r4, [r8] //Store the new head location in data

str r1,[r4+256] //Draw new head

b moveSnake

body: dat 0 //Initial front of queue (screen address for tail)

Make the modifications and run the simulation, stopping it after the snake has moved down one whole line. Take a snapshot of the simulation, showing that more of the memory now has values in it.

Even when the snake remains at just 2 segments, the memory gets used up as it moves, with the pointers for head and tail just pointing to new memory locations. Wouldn’t it be more efficient to keep the tail of the snake at the start of the body data, and just change the values as it moves, extending the length of the data only when the snake grows? Yes, that would use less memory - at least initially. But as the snake grows you would spend more and more processing time moving each segment of data up one location - and as the snake gets longer this would take more and more effort, slowing down the pace of the game.

Take a snapshot of the whole simulation. Highlight the values of registers r1, r2, r3 & r4, and then highlight the locations in main memory that show the ‘live’ snake data.

Leave the simulation to run. Notice that before the snake reaches the apple it stops and an error is shown. What is the error and why has it occurred? Hint: look at the memory on the right hand side of the simulation.

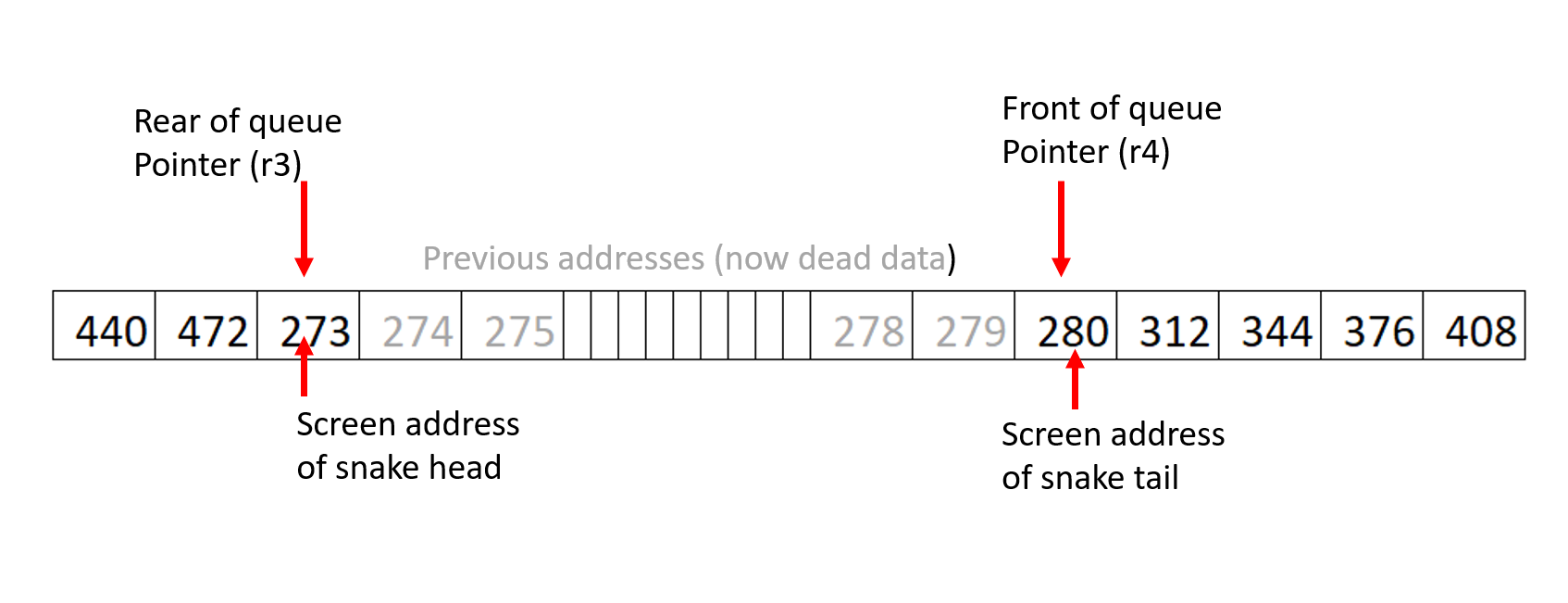
# Story 8: Fix memory overflow error

## Requirement

In the previous story we implemented a *linear queue* data structure in memory to hold the locations of the snake segments. The problem was that the linear queue quickly uses up all the available memory.

## Algorithm & techniques

We need to convert our *linear* queue into *a circular queue*. When either the front or rear pointers reaches the end of the available data area, it should be looped back to the start of the data area. Thus the ‘active’ data (representing the snake) will move through the memory repeatedly.



If the snake eventually grows long enough to fill the whole allocated data area then we will have to terminate the game - but the player will have beaten the game at that point. Meantime, that’s some way off.

(no changes before this)

moveTail:

ldr r0, [r7]

str r2,[r0+256] //Reset tail to Background

add r7,r7,#1 //Increment the tail pointer (for use next cycle)

cmp r7,#200 //Check pointer is still within memory

blt moveHead

mov r7, #body //If not loop pointer back to start of body data

moveHead:

add r8,r8,#1 //Increment the head pointer

cmp r8,#200 //Check pointer is still within memory

blt updatePointer

mov r8, #body //If not loop pointer back to start of body data

updatePointer:

str r4, [r8] //Store the new head location in data

str r1,[r4+256] //Draw new head

b moveSnake

body: dat 0 //body segment pointers extend from here to end of memory (addr 199)

Implement these changes. Run the simulation and let it run until the snake has eaten the apple, and then stop the simulation. By this point the circular queue holding the snake data (3 segments) should have looped back to the beginning.

Take a snapshot of the whole simulation. Highlight the values of registers r1, r2, r3 & r4, and then highlight the locations in main memory that show the ‘live’ snake data.

Note - you will still get an error when the snake eventually reaches the bottom-right corner of the screen. We will fix this later.

# Story 9: Change direction

## Requirement

Having refactored we can now return to what we attempted, but had to abandon in story 6:

*“When the S key is pressed, switch to moving downwards. This snake should continue downwards until its head reaches the bottom of the screen.”*

## Algorithm & techniques

We will need to read the keyboard to detect a key press, but without waiting for a key (as you would in a regular input field). The simulation provides an option for doing this:

inp r9,4

Will read the last key pressed into register 9. If no key has been pressed register 9 will contain zero.

## Code changes

defineRegisters: Initialise r9 at bottom of this section

mov r9, #0 //ASCII value of last key pressed

moveSnake:

str r6,[r5+256] //Draw Apple each cycle, in case it is on snake

inp r9,4 //Read the last key pressed (but don’t wait for one)

cmp r9,#83 //S key

beq down

right:

add r4,r4,#1 //Adding 1 to location moves right

b reDraw // Unconditional branch

down:

add r4,r4,#32 //...32 moves down one row on screen

reDraw:

~~add r4,r4,#1~~ DELETE THIS LINE

cmp r4,r5 //If the head is in same location as apple...

beq moveHead //...Skip updating the tail, to make snake grow

moveTail: (no changes after this)

Make the changes and paste in a screenshot showing the snake moving downwards.

# Story 10: Move in all four directions

## Requirement

Following the convention for many keyboard-driven video games, we will make the keys W,A,S,D correspond to changing the direction to Up, Left, Down, Right respectively.

## Algorithm & techniques

This is just a matter of following the same pattern we used to add the downwards switch, and also making sure that we handle right-wards movement in a consistent fashion.

## Code changes

no changes before this

moveSnake:

str r6,[r5+256] //Draw Apple each cycle, in case it is on snake

inp r9,4

cmp r9,#87 //W key

beq up

cmp r9,#65 //A key

beq left

cmp r9,#83 //S key

beq down

cmp r9,#68 //D key

beq right

right:

add r4,r4,#1 //Adding 1 to location moves right

b reDraw

down:

add r4,r4,#32 //32 moves down one row on screen

b reDraw

up:

sub r4,r4,#32 //-32 moves up one row on screen

b reDraw

left:

sub r4,r4,#1 //-1 moves left

b reDraw

reDraw: No changes after this

Make the changes and paste in a screenshot showing the snake moving upwards from the starting position

# Story 11: When apple is eaten, re-draw in a new, random location

## Requirement

Now that we can control the movement of the snake, instead of being in a fixed location the apple should be created at a random position on the screen - but never on the head of the snake.

## Algorithm & techniques

In the simulation, inp rN,8 will generate a random 8-bit number and put it in register N.

## Code changes

defineRegisters:

mov r1,#0x008844 //Snake colour (green)

mov r2,#0xffffff //Background colour (white)

mov r3, #271 //Tail position, initialised

mov r4, #272 //Head position, initialised

mov r5, #0 //Apple position

mov r6, #0xff8800 //Apple colour

mov r7, #body //Pointer front of queue

add r8,r7,#1 //Pointer to head address in body data

mov r9, #0 //ASCII value of last key pressed

mov r10, #767 //Constant representing the size of screen memory

mov r11, #1023 //Constant

InitialisePointers:

str r3, [r7] //r4 points to the tail address

str r4, [r8] //r3 points to the head address

drawSnake:

str r1,[r3+256] //Tail

str r1,[r4+256] //Head

createApple:

inp r5,8 // gets a random 32 bit pattern

and r5,r5,r11 // r11 has 1023

cmp r5,r10 // r10 has 767

bgt createApple // restrict random range

cmp r5,r4 // Make sure apples is not located on head of snake

beq createApple

# Story 12: When the apple is eaten, create a new one in a new position

## Requirement

At present, when the snake eats the apple, the snake grows but the apple is re-drawn in the same location – for the snake to eat again. Instead we want the eaten apple to disappear and a new one to appear, elsewhere, in a random location.

## Algorithm & techniques

We can just compare the location that the head has just moved (held in r4) against the location of the apple (held in r5). If they are the same then instead of looping back to where we re-draw the apple (moveSnake), we loop back to the routine that places the apple in a new random location (createApple).

## Code changes

updatePointer:

str r4, [r8] //Store the new head location in data

str r1,[r4+256] //Draw new head

cmp r4, r5 //Check again if the apple was eaten this cycle

beq createApple //If so, loop back to creating the apple

b moveSnake //Otherwise just repeat the move cycle

# Story 13: Ignore non WASD keys

## Requirement

The user might accidentally hit a wrong key, or two keys together (which, depending on how your keyboard is configured, might produce a wrong value). If the user hits anything other than W,A,S, or D, it should be ignored and the snake continue moving in the same direction.

## Technique

If the last key hit (held in r9) does not match any of the four allowed keys, then copy the previous direction of movement (held in r12) into r9 then go back to the start of the key comparisons in order to then branch to the routine handling movement in that direction.

## Code changes

defineRegisters: Define & initialise r12 at the end of this section

mov r12, #68 //Current direction of movement, initially 'right'

moveSnake:

str r6,[r5+256] //Draw Apple each cycle, in case it is on snake

inp r9,4

switchOnKey:

cmp r9,#87 //W key

beq up

cmp r9,#65 //A key

beq left

cmp r9,#83 //S key

beq down

cmp r9,#68 //D key

beq right

mov r9, r12 //If not any of the recognised keys, use prev direction

b switchOnKey //and re-run the switch on key

right:

Then no changes until …

reDraw:

mov r12,r9 //Update current direction with latest key

cmp r4,r5 //If the head is in same location as apple...

beq moveHead //...Skip updating the tail, to make snake grow

# Story 14: Halt the game if snake hits the edge

## Requirement

At present the snake can wrap around the left and right edges of the screen, but will generate a memory address error if it reaches the top or bottom edge.

Instead, if the snake hits any of the four edges, the game should simply halt.

## Algorithm & techniques

Detecting the top and bottom of the screen is simple: if the head position (held in r4) is less than zero, then it has moved off the top, and if greater than 767 (a constant already held in r10), then it has moved off the bottom.

To detect right edge, we can take advantage of the fact that the screen is exactly 32 pixels wide, the first row being locations 0-31. If, while moving right, the head location enters location 32 or any subsequent multiple of 32, it means it is moving over the right-hand edge. To detect if the address is a multiple of 32 we can perform a bitwise AND with 31 (0111111 in binary) and compare the result with 0 - in other words identify when the six lowest binary digits of the head address are all zero.

For the left edge we want to identify when the six lowest binary digits of the head address are all 1 - so AND with 31 and compare the result to 31.

## Code changes

right:

add r4,r4,#1 //Adding 1 to location moves right

and r0,r4,#31

cmp r0,#0

beq gameOver

b reDraw

down:

add r4,r4,#32 //32 moves down one row on screen

cmp r4,r10

bgt gameOver

b reDraw

up:

sub r4,r4,#32 //-32 moves up one row on screen

cmp r4,#0

blt gameOver

b reDraw

left:

sub r4,r4,#1 //-1 moves left

and r0,r4,#31

cmp r0,#31

beq gameOver

b reDraw

gameOver: Add this at the end of the program, just before the body: is declared

halt //To stop program execution running into data area

body: dat 0 //body segment pointers extend from here to end of memory (addr 199)

# Story 15: Ignore reverses in direction

## Requirement

If the snake is travelling right, say, then it should not be possible to switch immediately to travelling left, without first moving at least one pixel up or down. All such attempted reversals of direction should be ignored.

## Algorithm & techniques

After reading the most recent key and branching to the code that implements the intended direction, we can first compare that key press against the direction the snake was already travelling (held in r12) and ignore it if they represent opposite directions, setting the current direction to the previous direction. (We already have code to this if an unrecognised key is pressed, so we can re-use that).

switchOnKey:

cmp r9,#87 //W key

beq up

cmp r9,#65 //A key

beq left

cmp r9,#83 //S key

beq down

cmp r9,#68 //D key

beq right

dontChangeDirection:

mov r9, r12 //If not any of the recognised keys, set r9 to prev direction

b switchOnKey //and re-run the switch on keys

right:

cmp r12,#65

beq dontChangeDirection

add r4,r4,#1 //Adding 1 to location moves right

and r0,r4,#31

cmp r0,#0

beq gameOver

b reDraw

down:

cmp r12,#87

beq dontChangeDirection

add r4,r4,#32 //32 moves down one row on screen

cmp r4,r10

bgt gameOver

b reDraw

up:

cmp r12,#83

beq dontChangeDirection

sub r4,r4,#32 //-32 moves up one row on screen

cmp r4,#0

blt gameOver

b reDraw

left:

cmp r12,#68

beq dontChangeDirection

sub r4,r4,#1 //-1 moves left

and r0,r4,#31

cmp r0,#31

beq gameOver

# Story 16: Halt game if snake hits itself

## Requirement

At present the snake can cross its own path. However, after the tail passes the same point the snake will end up with a gap in it - still functioning correctly, but looking very odd.

Before proceeding with this story, paste in a partical screenshot showing the snake having crossed its own path and with a gap in the snake.

To be more consistent with other versions of the snake game, the snake should not be allowed to cross its own path - letting it do so ends the game.

## Algorithm & techniques

We can detect the snake crossing its own path by reading the colour of the screen location that the head has moved to (before any re-drawing). If that colour is the colour of the snake then the head is about to cross its own body.

## Code changes

updatePointer:

str r4, [r8] //Store the new head location in data

checkForCrossing: // check if the snake has hit itself

ldr r0,[r4+256] // read, from screen, colour of pixel head is moving to

cmp r0,r1 //If it is snake colour …

beq gameOver

str r1,[r4+256] //Draw new head

cmp r4, r5 //Check again if the apple was eaten this cycle

beq createApple //If so, loop back to creating the apple

b moveSnake //Otherwise just repeat the move cycle

# Story 17: Check for max length of snake

## Requirement

If you are skilled enough you can get the snake to eat sufficient apples that storing all the segments of the snake would take more than the available memory. Rather than producing an error, the game should halt at this point and a message printed in the output window: *You win*.

## Algorithm & techniques

We can detect when the snake memory (the circular queue data structure) is full when the rear of queue pointer catches up with the front of queue pointer.

To print a very short message in the output window, the simulator provides us with an option to use the out instruction to print ASCII-encoded text specified at a particular address.

## Code changes

checkForCrossing: // check if the snake has hit itself

ldr r0,[r4+256] // read, from screen, colour of pixel head is moving to

cmp r0,r1 //If it is snake colour …

beq gameOver

checkForMaxLength:

cmp r7,r8 //If front has caught up with rear, then body data full

beq gameWin

str r1,[r4+256] //Draw new head

cmp r4, r5 //Check again if the apple was eaten this cycle

beq createApple //If so, loop back to creating the apple

b moveSnake //Otherwise just repeat the move cycle

gameWin:

mov r0, #message //Load the starting address of the ASCII message

out r0,8 //Write message as text into output window

gameOver:

halt //To stop program execution running into data area

message: //ASCII encoding of 'You win' message (in reverse)

dat 0x20756f59 //' uoY'

dat 0x006e6977 //'niw'

body: dat 0 //Initial front of queue (screen address for tail)